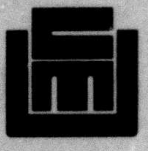
CMU-CS-82-141

Incremental Expression Parsing for Syntax-Directed Editors

Gail E. Kaiser*
Elaine Kant+

27 October 1982

DEPARTMENT
of
COMPUTER SCIENCE



Carnegie-Mellon University

CMU-CS-82-141

Incremental Expression Parsing for Syntax-Directed Editors

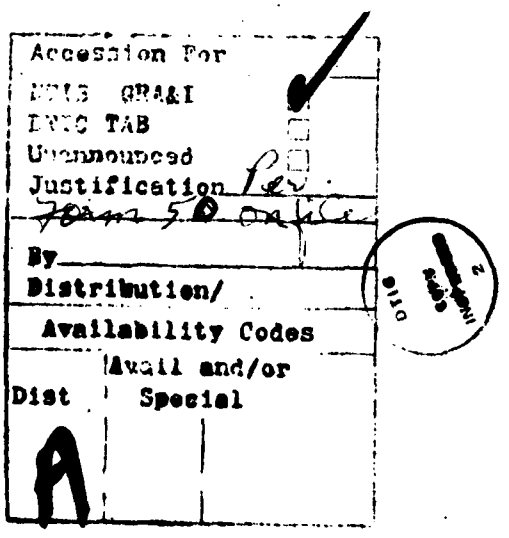
Gail E. Kaiser*
Elaine Kant+

27 October 1982

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, PA 15213

*Gail E. Kaiser is supported by the Fannie and John Hertz Foundation. Research on Gandalf is sponsored in part by the Software Engineering Division of CENTACS/CORADCOM, Fort Monmouth, NJ. +Elaine Kant is supported in part by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory Under Contract F33615-78-C-1551. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Hertz Foundation, the Defense Advanced Research Projects Agency or the US Government.

Table of Contents



List of Figures

Abstract

This document describes an algorithm for incremental parsing of expressions in the context of syntax-directed editors for programming languages. Since a syntax-directed editor represents programs as trees and statements and expressions as nodes in trees, making minor modifications in an expression can be difficult. Consider, for example, changing a " + " operator to a " * " operator or adding a short subexpression at a syntactically but not structurally correct position, such as inserting ") * (d" at the # mark in "(a + b # + c)". To make these changes in a typical syntax-directed editor, the user must understand the tree structure and type a number of tree-oriented construction and manipulation commands. This document describes an algorithm that allows the user to think in terms of the syntax of the expression as it is displayed on the screen (in infix notation) rather than in terms of its internal representation (which is effectively prefix), while maintaining the benefits of syntax-directed editing. The time and space complexities of the modifications for each new token are linear in the depth of the syntax tree.

1. Introduction

This paper describes an algorithm for incremental parsing of expressions in the context of syntax-directed editors for programming languages. The algorithm currently handles expressions involving only binary infix operators, but could be extended to n-ary operators and function calls and to statements for languages that can be handled by a precedence parser. Our algorithm differs from other incremental parsing algorithms in that it does not require the overhead of modifying a standard parser or maintaining a parser stack or any data structure other than the syntax tree. Instead, the algorithm applies tree transformations to maintain a correct syntax tree. The algorithm is table-driven and language-independent. Based on the user's command (to insert or delete an operator, for example) and the current state of the internal syntax, the relevant members of a small set of tree transformations are selected. Thus, after a token is inserted or deleted, transformations are automatically applied to restore the syntax tree to a correct state in terms of its precedence relationships and parentheses balancing.

Chapter 2 of this paper describes syntax-directed editing and provides motivation for our work. Chapter 3 describes how the user interacts with the editing environment. Chapter 4 describes the algorithm and gives several examples of its operation. Chapter 5 describes in more detail the transformations performed by the algorithm. The final chapter describes possible extensions to non-binary operators and statements and discusses our implementation of the algorithm. The appendix includes a summary of the precedence relations and the text of the full algorithm, minus those transformations best depicted by illustrations in Chapter 5.

2. Syntax-Directed Editing

A *syntax-directed* editor is a language-oriented editor in which programs are constructed and modified according to the syntactic constructs of the language rather than by editing characters and lines. A program is maintained internally as a syntax tree rather than as text. The text displayed on the screen is generated as needed by *unparsing* the internal representation.[1]

Besides the program editor, the same syntax tree may be used by other tools in a software development environment, such as semantic analyzers, intermodule consistency checkers, code generators, source-level debuggers and program transformation tools. The syntax tree, perhaps augmented with additional information, provides a uniform internal representation for these tools. In the case of the Gandalf environment [4] and ISDS [1], the syntax-directed editor also provides a uniform user interface for the integrated environment. The syntax-directed editor replaces both the text editor and parser components of the standard programming cycle.

In the external textual representation displayed on the screen of a typical syntax-directed editor, a *cursor* marks the node that will be the target of the next editing operation; the cursor is often displayed by highlighting the entire subtree rooted at the current node. The user enters a program by repeatedly selecting the *template* for the desired statement from the menu of templates legal at the current cursor position. The editor constructs a node for the selected statement, inserts it in the program syntax tree at the cursor position and displays the template on the terminal screen.

A template includes the *concrete* syntax, or "syntactic sugar", of the selected statement and leaves place holders (*meta* nodes) for the components specified by its *abstract* syntax. For example, the user might select the while statement by typing "w" and the editor would respond by displaying the while statement on the screen and positioning the cursor (denoted in this paper by underlining) at the first component. A meta node, which is a component of a template that has not yet been filled in, is displayed as *$type*, where *type* is the class of nodes that can legally be inserted at that component.

```
while $expression
    do $statement;
```

Expressions, as well as statements, may be entered by selecting templates. This method of expression entry is used in the ALOE system [6, 7]. For example, the user might select the less-than expression by typing "<."

---

[1] Some syntax-directed editors maintain a text buffer, with links between the text and the corresponding nodes in the syntax tree, and display the text buffer rather than unparsing the tree.

```
while $expression < $expression
    do $statement;
```

Variables and literals are entered by selecting the appropriate special template and then typing the variable name or literal value, respectively.

Syntax-directed editing applied to expressions is not without problems. First, expressions must be entered in (approximately) prefix notation rather than infix form even though the expression is displayed in infix form. This representation seems unnatural to many users. Another problem is that it is awkward to modify expressions. For example, to change "a + b * c" into "(a + b) * c", the user must either delete the original subtree and enter the desired subtree from scratch or perform a tedious series of **clip**, **delete**, and **insert** operations that require the user to understand the internal structure of the syntax tree.

One alternative exemplified by the Cornell Program Synthesizer [10] is to enter expressions as strings. When the user signals that an expression is complete, it is parsed and the resulting parse structure is inserted into the syntax tree. This allows the user to enter the expression in a natural way while maintaining most of the advantages of syntax-directed editing. This approach requires a complete reparsing of expressions that are modified in any way, which is relatively slow for large expressions.

A third alternative is to enter expressions (or statements) as strings that are *incrementally parsed* on entry. As the user enters the expression, the available input is parsed and each node is constructed as soon as possible. In Wegman's algorithm for the IBM PDE1L system [8, 11, 13] and Ghezzi and Mandrioli's algorithm [2, 3], the parser state is saved for each node in the syntax tree. Modifications are performed by entering the new string representation of the desired subexpression, restarting the parse before the first new token and continuing the parse to the end of the new subexpression. The user does not have to re-enter entire expressions or perform unnecessary modifications to the syntax tree.

Morris and Schwartz [9] have taken a slightly different approach to incremental parsing. Their algorithm maintains a series of parse trees for each expression, where a parse tree is split whenever a modification occurs in the text it covers. The trees are patched back together using an extension of traditional LL(1) parsing.

We have also taken an incremental approach. The differences between our approach and these standard incremental parsing approaches are:

1. We do not require the overhead of permanently maintaining a parser stack in addition to the syntax tree. Our algorithm does not require any data structures other than the syntax tree and its symbol table, both of which are already provided by the syntax-directed editor.[2]

2. We do not modify or extend a traditional shift-reduce or other type of parser that involves matching productions. Instead, we use a small set of tree transformations that are applied automatically as an expression is created and modified to keep the associated syntax tree in correct form with respect to precedence relationships and parentheses balancing.

3. Changes are reflected in the syntax tree immediately as they are typed in and converted to tokens by a lexical analyzer. (The modified expression is displayed on the next screen refresh.) Our algorithm does not produce a text representation of the program to be modified, perhaps in several places, and then partially reparsed so the tree can be updated; the text appears only on the display and is not stored (except on special command for such purposes as producing hardcopy).

4. Our algorithm does not necessarily fail on syntactically incorrect input. In some cases, new meta nodes or *empty operators* are inserted automatically to ensure the structural integrity of the syntax tree. The user can fill in the missing subexpressions later.

---

[2] However, a threaded list of tokens can be maintained to reduce the costs of cursor movement and unparsing.

3. The User View of the Editing Environment

In this chapter we describe the aspects of the user interface (of a hypothetical syntax-directed editor) that are relevant to our algorithm, including the cursor and the primitive operations provided.

The user sees the infix form of an expression displayed on the screen with a cursor highlighting the current editing position (which corresponds to some node in the syntax tree). The editing operations available to the user are:

- **construct** -- construct part of syntax tree;
- **delete** -- delete part of syntax tree;
- **clip** -- copy part of syntax tree to a buffer;
- **insert** -- insert contents of buffer into syntax tree;
- **replace** -- replace part of tree;

Other desirable editing operations that are not discussed in this document include **swap, undelete, mark and move.** The cursor movement commands are:

- **in (or down);**
- **out (or up);**
- **next (or right);**
- **previous (or left);**
- **root (or top).**

How these operations actually modify the syntax tree and the cursor depends on the current editing *mode*. Some of these commands are available only in some of the editing modes. There are three editing modes: node, token and character. The **mode** command is provided to switch modes. Node mode is the standard (and sometimes only) editing mode of most syntax-directed editors and character mode is related to the standard editing mode of most text editors. In the algorithm presented here, we are primarily interested in token mode.

3.1 Node Mode

Node mode is the normal editing mode of a syntax-directed editor. The user **constructs** a syntax tree by repeatedly selecting a *template* to be inserted at the current cursor position. The editor checks the legality of the selected template at the current position, constructs the node corresponding to the template, moves the cursor to the first empty component (*meta node*) of the template and displays the part of the syntax tree surrounding the new template on the screen. The cursor highlights the entire subtree rooted at the current node.

In this mode, **in** moves the cursor to the first child of the current node, **out** moves to the parent, **next** moves to the next sibling and **previous** moves to the previous sibling. The **root** command moves the cursor to the root node of the syntax tree. The **delete** command deletes the entire subtree rooted at the current node (the current cursor position). Similarly, **clip** copies a subtree to a buffer and **insert** replaces a meta node with the root of the subtree from a buffer. **Replace** replaces the selected subtree with the subtree from a buffer; it corresponds in node mode to a sequence of more primitive commands. In contrast to standard syntax trees, our algorithm requires the explicit presence of these parentheses in case the user later edits the expression in token mode, as explained below.

Although node mode is the most common editing mode in syntax-directed editors, it is only peripherally related to the algorithm described in this paper. We present it as a contrast to token mode, the primary editing mode relevant to this paper, and because tree modifications performed in token mode must be compatible with prior and later modifications made in node mode. Each of the different modes supports a slightly different type of modification of the syntax tree and is more convenient than the others in certain situations.

3.2 Token Mode

In token mode, each operator, terminal, open parenthesis or close parenthesis is considered to be a *token*. We assume that the tokens themselves are recognized by a table-driven lexical analyzer provided by the syntax-directed editor. The portion of a syntax tree representing an expression is normally constructed by repeatedly adding operator, terminal and parenthesis tokens to the right of the current cursor position, which is some token in the expression (this mechanism is called the **construct after** command, although typically the user types only the new token without giving any explicit command). However, other editing operations allow the user great flexibility in modifying the expressions (or partial expressions) that have been entered. We use a slightly non-standard

terminology in this paper since we assume a syntax tree rather than a parse tree: a *terminal* is some terminal element of the programming language that can occur as a leaf in a syntax tree, such as a literal or identifier. Operators and parentheses are not considered terminals because these occur only as the types of nodes, not as leaves, in a syntax tree.

The cursor motion commands available in token mode are **root**, **right** and **left**. The **root** command moves the cursor to the root of the current expression; it does not move outside the expression to the statement and procedure level. The user can switch to node mode for movement outside the current expression. **Right** moves the cursor to the token immediately to the right in the left-to-right display of the expression; **left** moves the cursor to the token immediately to the left. The cursor highlights only the current token.

The editing operations provided in token mode are given below.

- **construct after**: Operator, terminal and parenthesis tokens are repeatedly added to the right of the current cursor position in the screen display. The user does not actually type the **construct after** command; instead, she simply types the new token when the cursor is at the desired position. Internally, the algorithm creates a new node, adds it to the syntax tree and restructures the tree to preserve precedence relationships and parentheses balancing. After the tree transformation is complete, the screen is updated with the cursor at the new token.

- **construct before**: A token is added to the left of the current cursor position. The user must give the **construct before** command explicitly and only one token may be added for each command invocation. This command is necessary for adding tokens to the beginning of an expression.

- **construct at**: The cursor must be at a meta node or an empty operator. The given token, if it is of the correct type, replaces the place holder and the expression tree is restructured as necessary.

- **delete**: Typing the **delete** command removes the current token and restructures the syntax tree as necessary. The cursor is moved to the token to the left of the deleted token, or to the right if there is no token to the left.

- **replace**: The current token is replaced with the given token, which must be of the same type (operator or terminal), and the tree is restructured. The cursor is displayed at the new token.

The **clip** and **insert** commands are not provided in token mode because the effects can be easily

provided by node mode commands. In addition, node mode should be used for moving entire subtrees.

3.3 Character Mode

Character mode is for editing the text of terminals and operators; both cursor movement and editing commands are character-oriented. Character mode commands usually result in changes to tokens that are then translated into token mode commands. Switching into and out of character mode can be handled automatically. Character mode is not discussed further in this paper.

4. The Algorithm

4.1 Overview

When editing expressions in token mode, the user normally adds characters at the end of the expression and occasionally makes corrections by deleting and inserting tokens or sequences of tokens in the middle of the expression. First we will look at a few examples of how the algorithm adds tokens to the end of an expression, then we will see how tokens are inserted and removed from arbitrary locations within an expression. Throughout this chapter, we use the term "insert" interchangeably with "construct." Here, both terms refer to the **construct** user command described in Chapter 3 rather than to the **insert** user command that inserts the root node of a buffer into a position in a program tree.

It is assumed that all operators are binary infix operators. Each type of operator has a right-precedence and a left-precedence. A higher precedence means that the operator binds more tightly than an operator with lower precedence. For example, "a + b * c" means "a + (b * c)" rather than "(a + b) * c" because "*" has higher precedence than "+." Left and right associativity is subsumed by left and right precedence relationships; for example, the right precedence of a left associative operator is greater than its left precedence. A table of the operator precedences used in the examples presented in this paper is given in the appendix. Possible extensions to the algorithm for non-binary operators, for function calls and for statements are discussed briefly in Chapter 6.

The user's editing commands are reduced to a small set of basic tree-manipulating procedures, including structural transformations and cursor location changes. However, the user applies the commands to the text display and need not be aware of the underlying syntax tree. Most of the transformations are introduced by example here first, then described more generally in Chapter 5. The rest of the full algorithm is given in pseudo-code in the appendix.

4.2 The Syntax Tree Representation

The syntax tree is represented in a straightforward tree data structure. The leaf nodes are either terminals typed in by the user or *meta* nodes, indicating that a terminal or subexpression is expected but has not yet been specified. Intermediate nodes represent subexpressions consisting of an operator or parenthesis (the exact type is stored at the node) and two operands. The operands are stored as the left child and the right child of the node (there is only a single child for parenthesis

nodes). In addition, each node contains a back pointer to its parent node. The internal representation also includes a cursor variable which points to the node representing the token highlighted on the user's screen. It appears to the user that the cursor points to a particular token, rather than the entire subtree rooted at the node representing that token as in node mode.[3]

To achieve syntactic correctness, meta nodes and empty operator nodes are inserted and deleted as needed. An *empty operator* is a place holder operator node that has at least one child that is not a meta node. The operator may not have been filled in yet or it may have been deleted. Meta nodes are displayed to the user as "$META" and empty operator nodes as "$OP." In addition, open parenthesis, close parenthesis, and matched pair parentheses nodes are represented separately.

For example, if the user types "a + (* b," which is syntactically incorrect, it is stored internally and displayed as "a + ($META * b," which is correct but incomplete. If the user had made a mistake in typing and wanted to correct it by deleting the "+" or "*" operator, the "$META" node would be deleted as well. The expression "a + ($META * b" with the "b" token highlighted is stored internally as illustrated below. The cursor is a pointer to the "b" node. (In this paper, the cursor position is indicated by underlining.)

Figure 4-1: Syntax tree for "a + ($META * b."

4.3 Syntactic Correctness

A structurally correct expression tree is always maintained internally. In such a tree, all binary operators have exactly two children, all parenthesis nodes have exactly one child and terminals and meta nodes have no children. As described above, meta nodes and empty operators may be needed to maintain correctness. Tokens to the right (left) of a token in the display correspond to nodes that are right (left) *ancestors* of the token's node or descendants of these right (left) ancestors or right (left) descendants of the node itself.

---

[3] In node mode, the entire subexpression would be highlighted, for example, by a surrounding rectangle in reverse video.

To determine the left and right ancestors of a node, perform the following: Start at the node. If it is the left child of its parent, then the parent is one of its right ancestors; if it is a right child, then the parent is a left ancestor. This is repeated, comparing the grandparent and parent of the start node similarly. Right and left ancestors are those ancestor nodes to the right or left, respectively, of the current node. For example, see Figure 4-2.

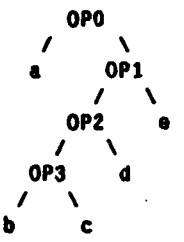


Figure 4-2: The right ancestors of "b" are OP1, OP2 and OP3. OP0 is a left ancestor.

A correct tree also satisfies the constraints that the right-precedence of every operator node is less than the left-precedence of its right child and that the left-precedence of every node is less than the right-precedence of its left child. The preceding constraint only applies when the children are operators; meta nodes, terminals and parenthesis nodes are also correct as children of any operator node.

Parenthesis balancing is an additional constraint. In a properly balanced tree, *all* tokens to the right of an open parenthesis (but before any matching close parenthesis) in the display should be descendants of the open parenthesis node (or the matched parenthesis node). Similarly, all tokens to the left of a close parenthesis (and after any matching open parenthesis) should be descendants of the close (or matched) parenthesis node. All nodes that are descendants of a matched parenthesis node should of course lie between the matching parentheses in the display. A properly balanced tree need not be complete: thus, some parenthesis nodes may be unmatched.

Consider for example, the correct syntax tree for "a + (b + c * d * e) * f ↑ g ↑ h" shown in Figure 4-3. Note that all the nodes that correspond to tokens appearing after the "d" in the display are *right ancestors* or their descendants and all those that correspond to tokens appearing before are *left ancestors* or their descendants (open parentheses are considered left ancestors and close parentheses are right ancestors).

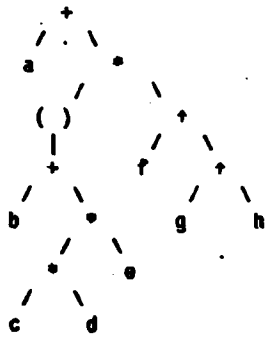


Figure 4-3: Correct syntax tree for "a + (b + c * d * e) * f ↑ g ↑ h"

The tokens between the parentheses in the display of Figure 4-3 all lie below the matched pair node and, within that subtree, the first "*" node lies below and to the left of the "+" node because it has greater precedence and appears after it. The second "*" node within the parentheses lies below and to the left of the first because the right precedence of "*" is greater than its left precedence. On the other hand, the first "↑" node lies above and to the left of the second one because the right precedence of "↑" is less than its left precedence. The precedence values of all operators used in the examples in this paper are given in a table in the appendix.

Suppose the close parenthesis were after the "f" rather than after the "e," giving "a + (b + c * d * e * f) ↑ g ↑ h." In this case, the tree would not be correctly balanced because the "*" and the "f" that would appear between the parentheses are not below the parenthesis node. The correct syntax tree is given in Figure 4-4.

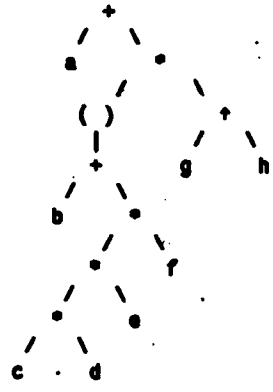


Figure 4-4: Correct syntax tree for "a + (b + c * d * e * f) ↑ g ↑ h"

4.4 Construct at End of Expression

In the typical case of adding tokens to the end of an expression (the construct after command),

the algorithm updates the syntax tree by inserting the new token at the cursor location,[4] then comparing the new token and its parent to make sure that precedence relations are satisfied and restructuring the tree if they are not. When adding an operator to the tree, a meta node representing the second operand is always added (the first operand is the node previously pointed to by the cursor). When a terminal is added to the end of an expression, the cursor must be pointing to a meta node, an operator node whose right child is a meta node that the terminal replaces or an open parenthesis node whose child is a meta node. In each case, the cursor is reset to highlight the new token.

For example, consider what happens when the user types in the expression "a + b * c."

1. The syntax tree is initialized to a single meta node with the cursor pointing to that node. (This meta node is actually a child of some statement in the program being constructed, but we will describe the algorithm as if the expression were itself the root node of the tree.)

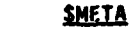


Figure 4-5: Root meta node.

2. When the "a" is encountered, the algorithm replaces the meta node with the identifier "a" and the cursor stays pointing at that node.

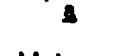


Figure 4-6: Meta node replaced by identifier "a" node.

3. After the " + " is typed, a new node is created. The new node is a " +" node whose left child is the "a" node and whose right child is a meta node. The cursor location is the " + " node. Since this node has no parent (in the expression), no further restructuring is considered.

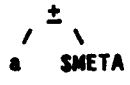


Figure 4-7: Addition of operator node.

4. When the "b" is typed, it fills the meta node that is the right child of the current cursor location (the " + " node) and the cursor is moved to the "b" node.

---

[4] The new token is always added at the bottom of the tree, and then migrated upwards to its correct position. If the token is a terminal it is added as a leaf node, otherwise it is added as the parent of an existing leaf node or a new meta node. If tokens were not added at the bottom of the tree, our algorithm would have to be more complicated to consider the possibility of migrating the new node downwards as well.

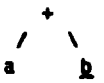


Figure 4-8: Child meta node replaced with identifier "b" node.

5. When the "*" is encountered, the same process happens as for the "+." A new node is created to replace the "b" node and becomes the right child of "b"'s parent. The new "*" node has the "b" node as a left child and a meta node as the right child. Since we now have an operator node with a parent, we must consider restructuring. However, in this case the left-precedence of the "*" operator is greater than the right-precedence of the "+" operator, that is, "*" binds more tightly than "+," so no changes are needed.

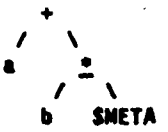


Figure 4-9: Addition of child operator node.

6. Finally, when the "c" is typed, it fills the meta node that is the right child of the "*" node (the current cursor location) and the cursor is moved to the "c" node.

Figure 4-10: Child meta node replaced with identifier "c" node.

This example illustrates two very simple transformations, *fill* and *nestleft*. Fill simply replaces a meta node by another node such as a terminal. This occurs in steps 2, 4 and 6. The nestleft transformation, which occurs in steps 3 and 5, replaces a given node by an operator node and the node becomes the left child of that operator. In any transformation, when a node "A" with parent "P" is replaced by a node "B," the parent and child links are updated so that "P" becomes the parent of "B" and so that "B" becomes the right (or left) child of the parent, just as "A" was. These transformations are illustrated in more detail in Sections 5.1 and 5.2.

Suppose instead the user types "a * b + c." The first four steps are analogous to those for the previous example. However, in the fifth step a reorganization of the tree is needed since the left-precedence of the new "+" node is less than the right-precedence of the parent "*" node. After the user types the "+," the new operator node is added as usual.

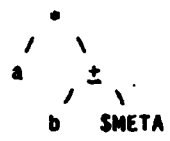


Figure 4-11: Addition of child operator node.

However, since the precedence constraints are not met, restructuring is needed. What happens is that the tree is transformed so that the new, lower precedence node (" + ") replaces the the higher precedence node (" * "), and the higher precedence node becomes the left child of the new operator. The left child of the new operator becomes the new right child of the higher precedence node. The cursor stays at the most recently added node, " + ." The result is:

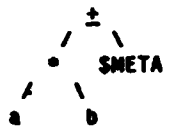


Figure 4-12: After restructuring.

The last step is that the "c" fills the meta node as before. (The actual **construct after** algorithm is given in pseudo-code in the appendix.)

The restructuring just described is a very basic transformation called the *twiddleright* transformation. There is also a *twiddleleft* transformation that is similar, but swaps the left child and its parent operator rather than the right child and its parent. These transformations are illustrated in Section 5.4. There are many transformations given in this paper that have both *xxx*left and *xxx*right forms. In general, the former involves the left child of an operator and the latter involves the right child. The two transformations are mirror-images of each other.

If the higher precedence node had a parent, it would be compared with the new node, now its child, in the same fashion. For example, suppose the user types "a - b ↑ c - d." The first six steps are analogous to those in the first example, since the left-precedence of the "↑" operator is greater than the right-precedence of the "-" operator. When the second "-" token is typed, the new operator node is created and a nestleft performed.

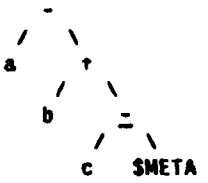


Figure 4-13: After nestleft.

Then the new operator node is compared to its parent. Since the right-precedence of "↑" is greater than the left-precedence of "-," a twiddleright is performed.

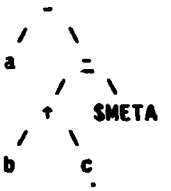


Figure 4-14: After first twiddleright.

The operator node is now compared to its new parent. Since the right-precedence of "-" is greater than its left-precedence, that is, "-" is left-associative, another twiddleright is performed.

Figure 4-15: After second twiddleright.

Since the new "-" node is now the root of the tree, the procedure halts. Finally, the "d" fills the meta node.

This whole process of repeated comparisons and twiddles (and some other transformations described later) is called the *rippleup* procedure. Whenever this process begins, the entire tree with the possible exception of the newly added operator satisfies the precedence constraints. The only possible violation the twiddle transformation creates (since it guarantees that the constraints between the new operator and its original parent are now correct) is that the precedence relationship between the new operator and its new parent may be incorrect. This is exactly what rippleup checks. The rippleup process continues until all precedence relationships are correct. This process is described in detail in Section 5; it is also given in pseudo-code in the appendix.

In the actual algorithm, an optimization is made to avoid doing the twiddle transformations one by one. Instead, the node at the bottom is checked in turn against a chain of operators all going in one direction (all left or all right ancestors) and then inserted only when the proper location is found according to the constraints or when the direction of a chain changes. This process is called the *macrotwiddle* transformation. An example of this is given later in Section 4.5.

Parenthesis balancing is similar in some ways to correcting operator precedence relationships. Inserting an open parenthesis at the end of an expression is quite simple. There are two cases: (1) the open parenthesis follows an operator or another open parenthesis and (2) the open parenthesis follows a terminal or a close parenthesis.

In the first case, the open parenthesis is nested onto the token immediately to its right in the display and this token becomes its child. The open parenthesis is then migrated up the expression tree to its correct position using the rippleup process: rippleup handles parentheses balancing as well as precedence constraints. The second case syntactically incorrect and triggers an error message. Syntax errors involving parentheses are not corrected because of the ambiguity with function calls and array references. For example, the user probably intends an identifier followed immediately by an open parenthesis to represent a function call rather than a binary expression with a missing operator.

Inserting a close parenthesis at the end of an expression is also simple and approximately symmetric to the open parenthesis case. The two cases are: (1) the close parenthesis follows a terminal or another close parenthesis and (2) the close parenthesis follows an operator or an open parenthesis. For the syntactically correct first case, the close parenthesis is simply nested onto the cursor (the terminal or close parenthesis), which becomes its child, and then rippled up the tree to its correct position. The second case is not permitted.

Suppose the user types in "a * (b + c * d)." The steps are as follows.

1. After "a *" has been typed, the syntax tree is:

Figure 4-16: Syntax tree after "a *" is typed.

2. When the "(" is typed, it nests onto the meta node, which is the token immediately to the right of the current cursor location (the "*" node). The cursor is moved to the new "(" node.

Figure 4-17: Nest the child meta node in an open parenthesis node.

3. When the "b" is encountered, it fills the meta node child of the open parenthesis and the cursor moves to the "b" node.

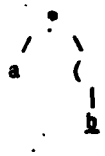


Figure 4-18: Meta node replaced by identifier "b" node.

4. Then "+ c * d" is typed and the algorithm continues as previously. The cursor is placed at the "d", the last token typed.

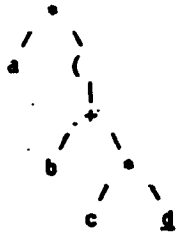


Figure 4-19: After "a * (b + c * d" is typed.

5. Finally, when the ")" is added, first a nestleft and then a rippleup is performed.

   a. First the parenthesis is nested onto the node indicated by the current cursor position.

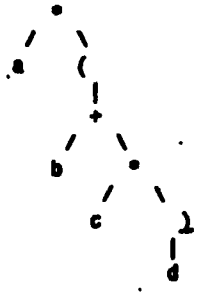


Figure 4-20: After nestleft.

   b. The rippleup process now compares the ")" parenthesis with its parent. Since the parent is an operator, a twiddleright is performed. Actually, the twiddle

transformations are not defined for parenthesis nodes. Instead, the macrotwiddleclose transformation is used for both this and the following step, to perform the equivalent of a series of twiddles.

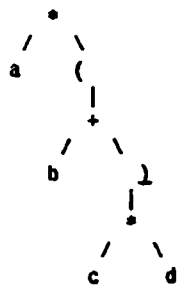


Figure 4-21: After first twiddleright.

c. It then compares the ")" parenthesis with its new parent. Since the parent (" + ") is again an operator, another twiddleright is performed (conceptually).

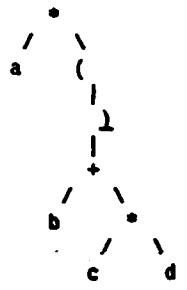


Figure 4-22: After second twiddleright.

d. Now the parentheses must be matched, using the *matchparens* transformation. This transformation folds the two parenthesis nodes into a single matching parentheses node. The cursor is moved to the new combined node with an internal notation that it is at the close parenthesis side. The rippleup is now complete. If the matching open parenthesis had instead been part of a matched parenthesis node, then rippleup would have been applied recursively to the newly unmatched close parenthesis node from which the open parenthesis was "stolen" and to the original child of the newly unmatched parenthesis node, since its migration up was previously halted by the matching parenthesis.

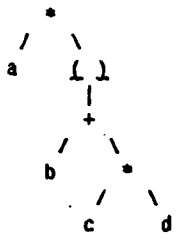


Figure 4-23: After matchparens.

The macrotwiddle family of transformations is described in detail in Section 5.6. Matchparens is described in Section 5.8.

Constructing at the end of an expression is really just a special case of constructing in the middle or at the beginning of an expression. It was described first so that the transformations would be easier to understand. In the next section we give some examples of inserting new tokens into the middle of an expression, then in the next chapter we discuss how the transformations are generalized.

4.5 Construct Within an Expression

Inserting tokens in the middle of an expression is more difficult than adding them at the end because all the tokens that occur after the inserted expression may change their position in the syntax tree. To avoid reparsing the entire remaining input, the new token is added at the bottom of the tree and the effect is migrated up using the rippleup process. The token is added at the bottom by finding the leaf node whose token immediately follows the current cursor location.

For example, to add a specified operator after a terminal (or close parenthesis), do a nestleft of the terminal (close parenthesis), then rippleup. Suppose the current expression is "a ↑ b * c * d + e," and suppose the user wants to insert " + g" after the "b." Then the user moves the cursor to the "b" and types " +." Before the construct after, the syntax tree is as follows.

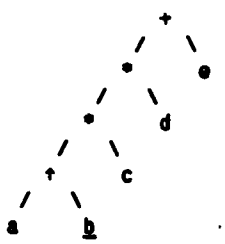


Figure 4-24: After moving the cursor to "b."

After nesting the new " + " operator, the tree is:

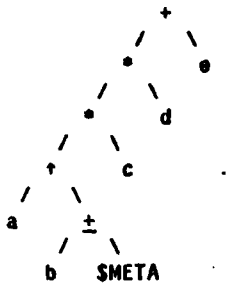


Figure 4-25: After a nestleft of the "b" into the operator " + ."

Next, the macrotwiddleright transformation is applied. This transformation looks at nodes that correspond to operators that appear to the left of the new operator in the display and restructures the tree if necessary. In the syntax tree, the operators to the left are found in the chain going upward from the parent of the new operator (in this case, only the single node "↑"). The search stops when a node is found whose left precedence is less than the right precedence of the new operator (or it stops at the root or when the chain stops going up to the left). The new node belongs below the stopping point. In this case, the node found is the grandparent node " * ."

As shown in Figure 4-25, the new operator node is then brought up to be the right child of this stopping node. Its old parent (the "↑" node) becomes the new node's left child, and the old left child becomes the right child of the old parent.

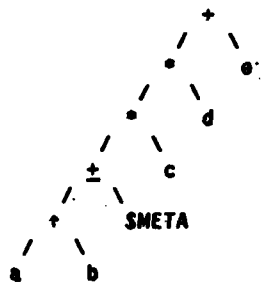


Figure 4-26: After macrotwiddleright.

Since the chain of ancestors is now to the right, the macrotwiddleleft transformation is applied. In this case, the stopping node is the root node " + ." The new operator node is then brought up to be the left child of this stopping node. All the nodes between the stopping node and the old parent are brought down to be the right children of the new operator, and the leftmost grandchild is set to be the meta node that was previously the right child of the new operator. The result is illustrated in Figure 4-26.

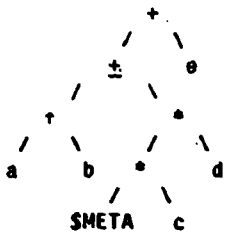


Figure 4-27: After macrotwiddleleft on the new " + " node.

To add a terminal after an operator (or open parenthesis), first find the right adjacent terminal. If it is a meta node, fill it with the terminal, otherwise do a nestleft of it into an empty operator.

Inserting parentheses in the middle of an expression is a bit more difficult since we need to find matching parentheses and bring them adjacent, which may change the pairing of other parentheses. If an open (close) parenthesis is added to the middle of an expression, it is nested above the following (preceding) terminal, then the rippleup process is applied until another parenthesis-type node is found. The special parenthesis transformations -- macrotwiddleopen/close and matchparens -- are used in the rippleup process.

Consider inserting an open parenthesis after the cursor in "(a * b + c) ↑ d."

1. First the open parenthesis is nested onto the "b" terminal node at the bottom of the tree.

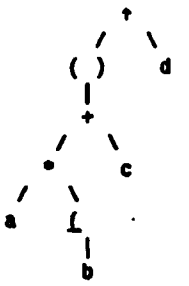


Figure 4-28: After nest of "("

2. Now the macrotwiddleopen transformation is applied to migrate the new open parenthesis up to its correct position and change the relative positions of the "*" and " + " operators.

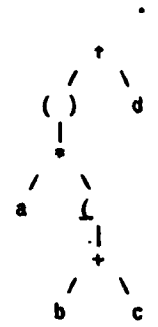


Figure 4-29: After macrotwiddleopen.

3. Now matchparens is applied to find the close parenthesis matching the new open parenthesis. This parenthesis is "stolen" from the matched parenthesis node higher in the tree.

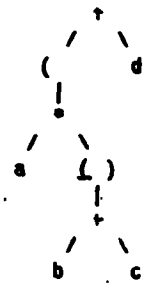


Figure 4-30: After parentheses are matched.

4. Since the close parenthesis was "stolen", the matchparens transformation now invokes the rippleup process twice, first on the newly unmatched parenthesis node and then on its original child (its child before the first rippleup was invoked). The first invocation migrates the open parenthesis to the root of the expression. The second rippleup twiddles the "*" and "↑" operators into their correct precedence relationship. The insertion of the new open parenthesis node is now complete.

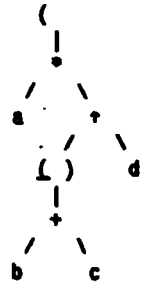


Figure 4-31: After matchparens.

4.6 Other Construct Commands

In addition to **construct after**, the **construct before** and **construct at** commands are also provided. In general, **construct before** is symmetric with **construct after**, since the token is inserted before the current cursor position rather than after it in the display. To perform a **construct at**, the cursor must be at a meta node or an empty operator. Typically, the new token is nested onto the meta node and then rippled up the expression tree. Similarly, a new operator token may replace an empty operator and be rippled up to its correct position. The algorithms for these two commands are given in the appendix.

4.7 Delete

The **delete** command causes the highlighted token to be deleted. It is not possible to delete a meta node or empty operator node unless it is adjacent (in the display) to an empty operator or a meta, respectively; otherwise, the expression would become syntactically incorrect. In addition, sometimes these nodes are automatically removed from the tree to maintain the syntax structure when another token is deleted. After the delete occurs, the tree is redisplayed with the cursor at the token to the left of the deleted token, if there is one, and to the right otherwise.

In the case where the highlighted token is an operator, the algorithm checks whether there is an adjacent meta node on either side of the operator. If so, it performs a *collapseleft* or *collapseright* transformation to remove both the operator and the meta node. Otherwise, it replaces the operator with an empty operator node.

Consider the expression "a * $META + b" with the cursor at the " + " node.

Figure 4-32: Syntax tree for "a * $META + b."

When the " + " token is deleted, both the "$META" and " + " nodes are removed using the *collapseright* transformation. First, the " + " operator node is replaced with its left child, the "*." Its right child is held aside.

Figure 4-33: Children of the deleted operator node.

Then the meta node is replaced by the operator's right child. If the "+" operator node had a parent, a rippleup would be performed.

Figure 4-34: After collapseright.

The algorithm is similar in the case of a terminal. If there is an adjacent empty operator, both are removed using a collapse transformation. Otherwise, the terminal is replaced with a meta node.

In the case of an open parenthesis, the parenthesis node is removed and the rippleup transformation is performed on its child to correct precedence relationships. In the case of the open half of matched parentheses, the same transformation is performed. In addition, the new close parenthesis is migrated up the tree using rippleup in order to balance parenthesis. If a matching parenthesis is not found, the close parenthesis migrates to the highest legal position in the tree. The delete of a close parenthesis or the close half of matched parentheses is symmetric.

The full delete algorithm is given in pseudo-code in the appendix.

4.8 Replace

The other editing operation available in token mode is replace, which is defined only for an operator token. The operator node is replaced with the new operator, retaining the same children. Then the rippleup transformation is applied to the operator node. If no changes are made, this indicates that the portion of the tree above the node is consistent. It may be the case, however, that the new operator has a higher precedence than one of its children, so the *rippledown* transformation is then applied to the operator node.

If the left child of the operator is itself an operator, the rippledown transformation compares the left-precedence of the operator to the right-precedence of the left child. If greater than, it performs a twiddleleft and continues the rippledown. The case for the right child is similar. A more detailed description of rippledown is given in Section 5.10; rippledown is also given in pseudo-code in the appendix.

For example, consider the expression "a * b + c + d" with the cursor at the leftmost " + " node.

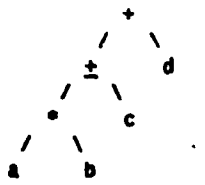


Figure 4-35: Syntax tree of "a * b + c + d."

The user gives the replace command and types "↑." Rippleup is applied first. The "↑" operator has a higher right-precedence than the left-precedence of " + ," so the rippleup fails.

Then rippledown is applied. Since the "↑" has a higher left-precedence than the right-precedence of the "*" operator, the twiddleleft transformation is applied and the "↑" node becomes the child of the "*" node, which in turn becomes the child of the remaining " + " node. Since all children of "↑" are now identifiers, rippledown halts.

Figure 4-36: Before and after rippledown.

5. Summary of the Transformations

This chapter describes each of the transformations performed by the algorithm and some constraints preserved by the transformations. First, we present the simple transformations: fill, nest, unnest and twiddle. Then we give the more global transformations: collapse, macrotwiddle and matchparens. Finally, the rippleup and rippledown processes are presented. The global transformations and these two processes invoke each other recursively. The rest of the full algorithm, (the part of the algorithm that selects among these transformations and applies them to the expression tree) is given in the appendix.

5.1 Fill

The *fill* transformation simply replaces a meta node with a new node denoting a terminal, parenthesis or operator.

Figure 5-1: Before and after filling $META with <new>.

If the new node is a parenthesis or an operator, the parentheses in the expression may no longer be balanced and the precedence relationships may no longer be consistent. There is a process called *rippleup*, described in Section 5.9, that calls other transformations to re-establish these relationships.

5.2 Nest

There are two flavors of *nest*: nestleft and nestright. Both forms of nest take two arguments, a node and a type. They create a new operator or parenthesis node of the given type and insert it between the given node and its parent. *Nestleft* makes the given node the left child of the new node and *nestright* makes it the right child of the new node. The other child is a meta node. If the new node is a parenthesis node, nestleft and nestright are identical and make the argument node the child of the new parenthesis node. In either type of nest, if the node was the left child of its parent, then the new node becomes the left child of the parent, and similarly in the case of right child.

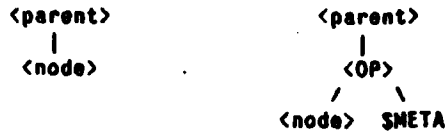


Figure 5-2: Before and after nestleft of <node> and <OP>.

Figure 5-3: Before and after nestright of <node> and <OP>.

Since nest creates a new operator or parenthesis node and inserts it into the tree between two other nodes, the relationships between the node and its new child (and further descendants) and between the node and its new parent (and further ancestors) may be incorrect.

In all cases where the nest transformation is used by the token mode algorithm, the argument <node> is a terminal; thus the relationship between it and the new <OP> node is correct and there are no further descendants. The rippleup process is used to correct the ancestor relationships of the new node.

5.3 Unnest

Unnest is the inverse of nest. *Unnestleft* deletes the given node and replaces it with its left child. *Unnestright* deletes the given node and replaces it with its right child. In both cases, the other child is assumed to be a meta node or otherwise undesired. These transformations are exactly the inverses of those shown in Figure 5-2 and Figure 5-3.

After the unnest transformation has replaced a node with its child, the relationships between the child and its ancestors may be incorrect and must be checked by the rippleup process. However, the relationships within the subtree rooted at the child remain consistent.

5.4 Twiddle

Now we come to the first interesting transformation. The *twiddle* transformation modifies the structure of the tree by changing the relative positions of three nodes. It does not construct or delete any nodes.

Twiddleleft changes the relative positions of an operator node and its left child, which must also be an operator. The third node involved is the right child of the left child. Let's call the operator node "OP1," its right child "N1," and the pair and N1's descendants "subtree 1." Call the left child "OP2," call its left child "N2," and call the pair and the descendants of N2 "subtree 2." The right child of OP2 is "N3," which together with its descendants is "subtree 3." The parent of the operator node is called "P."

In the twiddleleft transformation, the root of subtree 2 replaces the root of subtree 1 as the child of node P. The root of subtree 1 replaces the root of subtree 3 as the (right) child of subtree 2 (of OP2). The root of subtree 3 replaces the root of subtree 2 as the (left) child of subtree 1 (of OP1). In each case, the rest of the subtree is carried along with its root. Basically, the twiddleleft transformation changes the left association of the relevant operators to right without changing the order of the leaves.

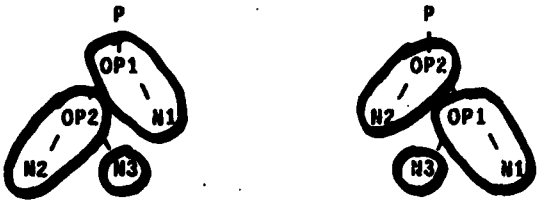


Figure 5-4: *Before and after twiddleleft applied to OP1.*

Each of the three subtrees described above is internally consistent at the time when the twiddleleft is applied, although the relationship between OP1 and OP2 is incorrect. The internal consistency is not changed by the transformation. However, after the twiddleleft is performed, OP2 may in turn be in the incorrect position relative to the parent node P, so further transformations may be necessary until OP2 has migrated up to its correct position. It is also possible that N3 and OP1 are not in the correct relationship. When twiddleleft is applied by rippleup from the bottom up, N3 is always either a terminal or a node that was adjacent to OP1 before a previous twiddle, so the relationship is actually correct. When twiddleleft is applied by rippledown, it is exactly the relationship between N3 and OP1 that is checked.

The twiddle transformation of course maintains the correct right-left positions of all nodes. For example, before the twiddleleft OP1 had right descendant N1, and left descendants N3, OP2 and N2. After the twiddleleft, N1 is still a right descendant, N3 is now a direct left descendant, OP2 is a left ancestor and N2 is a left descendant of a left ancestor. Similarly, OP2 is transformed from having right ancestor OP1 and right descendants N3 and N1 to having all three as right descendants and N2 remains a left ancestor. Note that the leaves are maintained in the original left-to-right order.

Twiddleright is the mirror image (and the inverse) of twiddleleft. It changes the relative positions of an operator node and its right child, which must also be an operator. The third node involved is the left child of the right child.

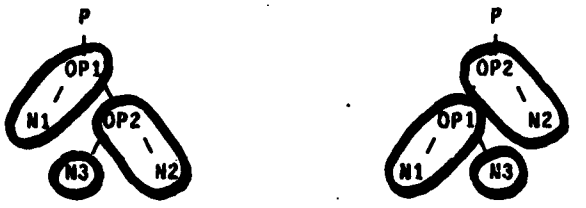


Figure 5-5: Before and after twiddleright applied to OP1.

5.5 Collapse

The two *collapse* transformations may be used when an operator or a terminal is deleted. If an empty operator and a meta node are adjacent in the display, the transformations remove both of them from the tree. *Collapseright* removes a meta node and its right adjacent empty operator while *collapseleft* handles a left adjacent empty operator.

Collapseright replaces the operator with its left child (which may or may not be the meta node, since the meta node may be a distant descendant) and then replaces the meta node with the operator's right child. If the meta was in fact the operator's left child, this has the effect of an unnestright transformation, the inverse of a nestright. Otherwise the effect is equivalent to performing a *rippledown* on the empty operator (the empty operator has a higher precedence than any other operator), which will make it the parent of the meta node, and then performing an unnestright. Finally, a rippleup must be performed on the former right child, which has replaced the operator, in order to restore precedence constraints.

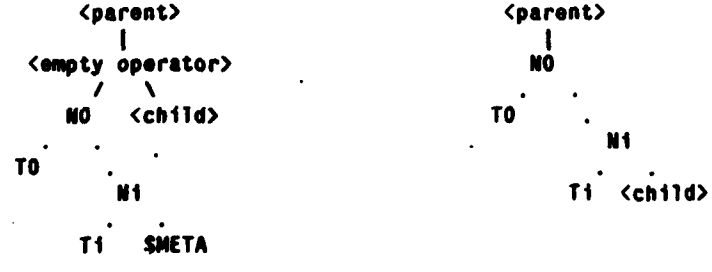


Figure 5-6: Before and after collapseright of <empty operator> and $META.

The collapseleft transformation is the mirror-image of collapseright.

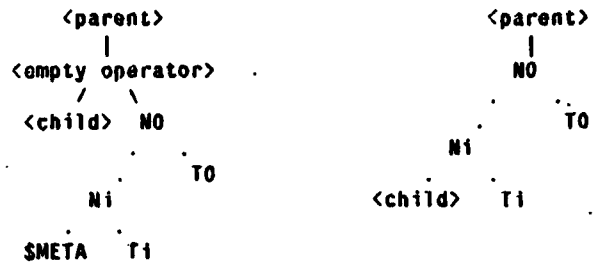


Figure 5-7: Before and after collapseleft of <empty operator> and $META.

5.6 Macrotwiddle for Operators

The *macrotwiddle* family of transformations is used by the rippleup transformation described in Section 5.9. Macrotwiddleleft and macrotwiddleright are similar to the two *twiddle* transformations in form, but may involve nodes that are not adjacent in the tree. They are equivalent to performing a series of twiddle transformations. Macrotwiddleopen and macrotwiddleclose enable parentheses balancing by rippling unmatched parentheses through the tree.

Macrotwiddleleft, like twiddleleft, is a transformation on three subtrees to correct inconsistencies in operator precedence relationships. Since macrotwiddleleft is called from within rippleup, the operator node in focus is analogous to "OP2" of the twiddle transformation. As before, we will call its left child "N2," and call the pair and the descendants of N2 "subtree 2." The right child of OP2 is "N3," which together with its descendants is "subtree 3." Call the parent of OP2 node "OP0." Now rather than simply comparing OP2 to its parent, we will look up the chain of operator ancestors until we find one whose left precedence is less than the right precedence of OP2, or until the chain reaches the root or stops going up to the right. Let's call the operator node at this stopping point "OP1," its right child "N1," and the pair and N1's descendants down to OP0 "subtree 1." It is possible that OP1 is the same node as OP0. The parent of OP1 is called "P."

As in the twiddleleft transformation, the root of subtree 2 (OP2) replaces the root of subtree 1 (OP1) as the child of node P. The root of subtree 1 (OP1) replaces the root of subtree 3 as the (right) child of OP2. The root of subtree 3 replaces the root of subtree 2 as the (left) child of the leftmost operator in subtree 1 (OP0). In each case, the rest of the subtree is carried along with its root.

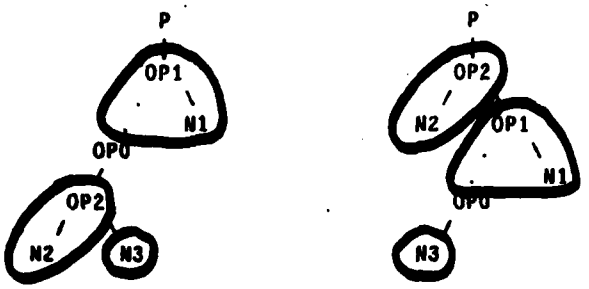


Figure 5-8: Before and after macrotwiddleleft of OP2.

Macrotwiddleright is the symmetrical transformation applied from rippleup when an operator is a right child of its parent operator.

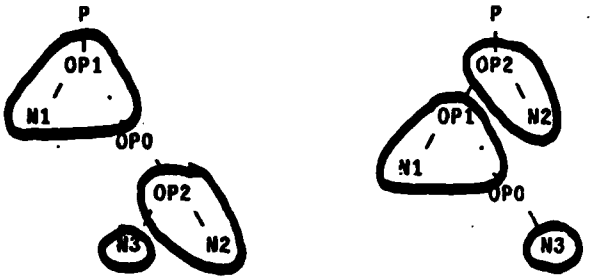


Figure 5-9: Before and after macrotwiddleright on OP2.

5.7 Macrotwiddle for Parentheses

Macrotwiddleopen and *macrotwiddleclose* are similar to the left and right macrotwiddle transformations but are used to restructure the tree to migrate parentheses to the proper locations. They too are called by the rippleup procedure (after matchparens has checked for any matches). These macrotwiddles do not check precedences along the chains since they assume those are already correct.

Macrotwiddleopen moves subtrees that are above an open parenthesis down below it. Like macrotwiddleleft, it is a transformation on three subtrees. However, the second subtree is more inclusive than the previous form, which included only operator OP2 and its left descendants. Here again we call the operator that is the first right ancestor in the upward chain "OP0." Its left child, which also is the leftmost ancestor of the open parenthesis, is called "OP2," and OP2's left child is called N2. Together, OP2, N2, all N2's descendants, plus all of OP2's right descendants down to and including the open parenthesis is called "subtree 2." The child of the open parenthesis is "N3," which together with its descendants is "subtree 3." Again, we look up the chain of ancestors of OP0 (but without checking precedences) until we find a parenthesis node, or until the chain reaches the root or stops going up to the right. Let's call the operator node at this stopping point "OP1," its right child "N1," and the pair and N1's descendants down to OP0 "subtree 1." It is possible that OP1 is the same node as OP0. The parent of OP1 is called "P."

The transformation is then to replace the root of subtree 1 (OP1) with the root of subtree 2 (OP2) as the child of node P. The root of subtree 1 (OP1) replaces the root of subtree 3 as the child of the open parenthesis. The root of subtree 3 replaces the root of subtree 2 as the (left) child of OP0.

The rippleup process is then applied again to the open parenthesis in its new location.

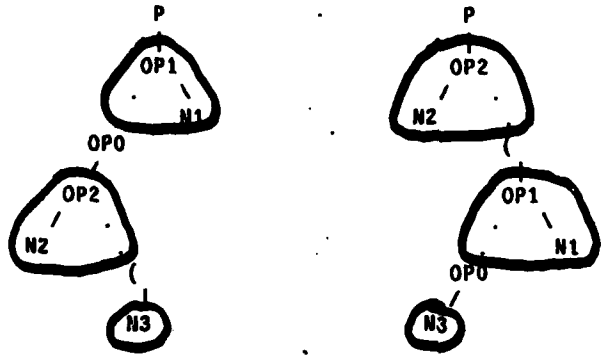


Figure 5-10: Before and after macrotwiddleopen.

Macrotwiddleclose is symmetric to macrotwiddleopen.

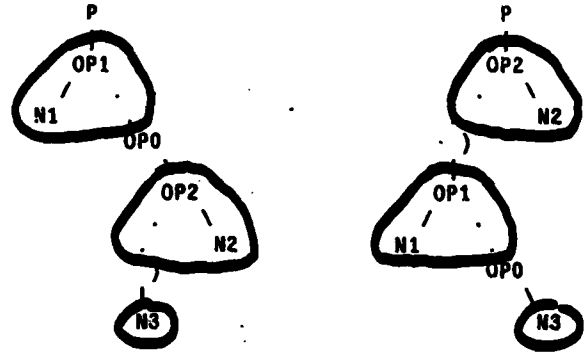


Figure 5-11: Before and after macrotwiddleclose.

5.8 Matchparens

The *matchparens* transformation is called by the rippleup process whenever the argument to rippleup is either an open or close parenthesis and there is a matching parenthesis close by. A newly added parenthesis is propagated through the tree by rippleup until it reaches either its correct position or the root of the expression. To check for a match of an open (close) parenthesis, trace up the left (right) ancestor links until a parenthesis, the root or a right (left) ancestor is found. If there is parenthesis that is a matched pair or a close (open) parenthesis, then a new match has been found. There are two cases:

1. The node is exactly the matching parenthesis. The two nodes are merged into one matched parentheses node.

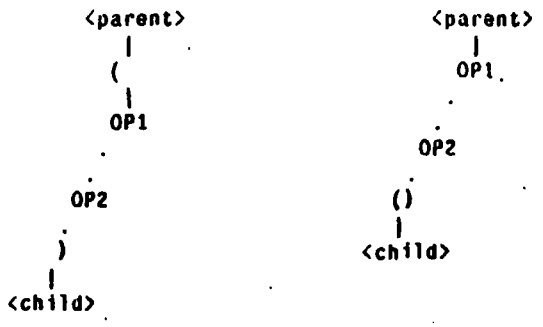


Figure 5-12: Before and after matchparens of ")" with "(."

After this transformation is performed, the rippleup process is applied to OP1 since the precedence relation between OP1 and <parent> may not be correct. Only parentheses balancing and precedence relationships at the previous position of the moved parenthesis may be made inconsistent by the matchparens transformation.

2. The parent is a matched parentheses. Then the node "steals" its matching parenthesis from its parent and the parent is changed to the complementary type of unmatched parenthesis.

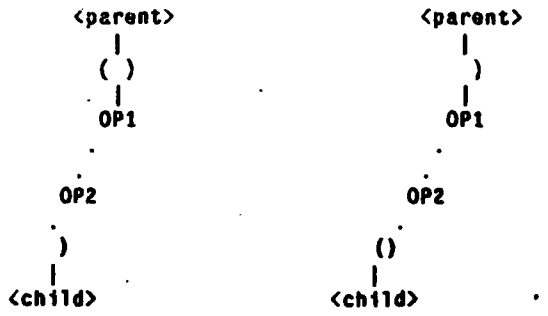


Figure 5-13: Before and after matchparens of ")" with "( )."

In this case, the upper parenthesis node is no longer balanced. Rippleup must be called to attempt to match it or migrate it to the correct location. Finally, rippleup must again be applied to OP1, the child of the newly unmatched parenthesis.

5.9 Rippleup

The rippleup routine is not really a transformation, but a process consisting of a sequence of transformations from the *macrotwiddle* family and matchparens (described in Sections 5.6, 5.7, and 5.8) that migrate an operator or a parenthesis up the tree. The process stops when the root of the tree is reached or when no additional changes are required or possible.

Suppose the node being rippled up is an operator. Conceptually, it is compared to its parent operator, and if the (right/left) precedence of the parent is greater than the (left/right)-precedence of the node, then a twiddle(left/right) transformation is performed and the process is repeated with the node at the higher location. As described in the twiddle transformation, if the subtrees already have the correct precedence relations, then only the newly connected nodes need be checked. Actually, macrotwiddle transformations are used as shortcuts. When the root or a matched parenthesis node is reached, rippleup stops.

If the node being rippled up is a parenthesis node, then the macrotwiddleopen and macrotwiddleclose transformations are used until either the root is reached or another parenthesis node is found. The matchparens transformation is used to balance parentheses and continue rippleup if a matched pair or exactly matching parenthesis node is found; if a parenthesis of the same type is found, then rippleup stops.

The rippleup process is presented in pseudo-code in the appendix.

5.10 Rippledown

The rippledown routine is also not a new transformation but rather a sequence of twiddle transformation applied repeatedly to a node until the node meets the precedence constraints (parentheses balancing is not affected). Rippledown is applied at an operator node when it is possible that the precedence relationships between the operator and its children may be incorrect. This happens in the **replace** user command, where an operator is replaced by a selected operator. The difference between rippleup and rippledown is that the former migrates a node up the tree when its relationships with its ancestors are incorrect; the latter migrates a node down the tree to correct its relationships with its descendants.

Rippledown starts by comparing the operator of the argument node to its left child, and then to its right child. If the left child is an operator node whose right-precedence is greater than the left-precedence of the argument, apply twiddleleft and then rippleup to the left child. If the left child is not migrated, then apply the symmetric process to the right child. If neither child moves, the new operator is now in the correct place and the transformation halts. However, if either child is migrated, then the process must be repeated recursively with the new children of the operator node. In the case where the left child was twiddled, the original right child is still the right child of the operator and is considered in the recursion.

The rippledown process is presented in pseudo-code in the appendix.

6. Conclusions

6.1 Related Work

As discussed in Chapter 2, our algorithm is similar in several respects to the incremental parsing algorithms developed by Wegman [11, 13, 12], Ghezzi and Mandrioli [2, 3] and Morris and Schwartz [9]. These algorithms involve modifications to traditional parsing algorithms.

The Morris and Schwartz method is to modify the LL(1) parse tables to treat entire subtrees as terminals in some cases. Their algorithm maintains a series of parse trees with links to the text buffer. The text may be modified provided it obeys the "language constraint" -- the text up to the token immediately preceding the cursor must be the legal beginning of a program. The text between two such discontinuities is covered by a separate parse tree. Whenever a command attempts to move the cursor beyond a discontinuity, the parse trees are patched back together using an extension of traditional LL(1) parsing. However, full reparsing may be necessary in many cases.

The Ghezzi and Mandrioli method is to modify a shift-reduce LR parser (they discuss possible extensions to LL parsers). The parser state is saved for each node in the entire syntax tree, with links between each node and its corresponding production on the stack. Modifications are performed by entering the new string representation of the desired subexpression and continuing the parse, using an extension of a traditional LR parsing algorithm, from the indexed point in the parse stack to the end of the new subexpression. The resulting incremental parser requires a considerable amount of storage to record the configurations entered by the parser at each step of the analysis. It assumes that the expression being parsed is syntactically correct, although it may be possible to remove this restriction.

The Wegman method is an improvement over Ghezzi and Mandrioli. It also involves modifying a shift-reduce LR(1) parser and permanent maintenance of the parser stack (which is stored in spaghetti form for maximum sharing). The resulting parser makes no more than the minimal number of changes required in the parse tree times a log factor of its height, whereas the earlier algorithm may take time proportional to completely reparsing the program in the worst case. The algorithm continues to operate in the face of syntactic errors in the modified portion of the program: it assumes that the last token(s) entered is (are) the source of the error; the tree is not modified except to note the position of these tokens in the display in relation to those represented in the legal syntax tree.

Our method is quite different in several respects. It does not involve the extension of standard

parsing techniques or the generation of a language-specific incremental parser from a grammar. The algorithm is language-independent and completely table-driven. The required tables include lexical analysis information for recognizing tokens and the set of legal operators and their left and right precedences.[5]

Our method does not involve additional data structures other than the syntax tree. In fact, we use the standard syntax trees supported by most syntax-directed editors, which are generally more compact than the parse trees generated by typical parsers [7]. We are considering, however, adding two fields to each node in an expression tree to link the tokens in the left-to-right and right-to-left order in which they appear in the display. The extra time required to create these links is negligible since the left and right tokens are generally found anyway as part of the algorithm, but once the links are created, this searching time would be drastically decreased (to a constant). The primary costs would be the extra space for storing the links and the difficulties involved in reading and writing the augmented syntax trees to a file.

Another advantage of our method is the removal of the necessity for maintaining a text buffer in parallel with the parse tree. In the other algorithms, changes are first made in the text representation and then the text buffer is partially reparsed to update the parse tree. Our algorithm supports the direct modification of the syntax tree without any intermediate changes in a text buffer. In fact, no text buffer exists: the text on the display screen is generated dynamically by *unparsing* the syntax tree; only the syntax tree is stored internally. Input is stored as a string only until it has been converted to a token by the lexical analyzer.

Like the Wegman algorithm, our method handles syntactically incorrect input. In cases involving only operators and terminals, our algorithm assumes the user intended to make these "mistakes" and automatically inserts visible meta nodes and/or empty operators for the user to fill in later. In cases involving parentheses, where there is an inherent ambiguity with function calls and array references, the incorrect parenthesis triggers an error message while the user is still in context.

The main disadvantage of our algorithm is that we have not yet worked out how to extend it to a more general class of languages than those handled by a precedence parser.

---

[5] We plan to augment the algorithm with a mechanism to handle a table of different types of brackets used as parentheses, plus a mechanism for handling those languages where the same brackets are used for other purposes, such as array references and function invocation.

6.2 Extensions

The algorithm should be extendible to other programming language constructs such as non-binary, prefix and postfix operators, function calls, array references, and statements (in at least those programming languages that can be handled by precedence parsers). If operators are unambiguous and the operands of arbitrary-arity operators are enclosed within delimiters, the extension to the algorithm is straightforward. Problems arise in handling operators such as "-," which may be either binary infix or unary prefix, but this problem could be handled by a table of such ambiguous operators and their properties. For example, suppose the expression is "a + b - c." In the usual binary infix case, deleting the "b" would cause it to be replaced by a meta node.

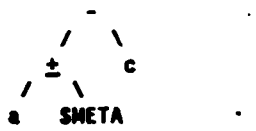


Figure 6-1: "-" as binary infix operator.

However, in the unary case, the "-" token could be recognized as a unary operator whose single child is the "c" identifier and the tree could be restructured to reflect this.

Figure 6-2: "-" as unary prefix operator.

The algorithm can also be extended to handle function calls and array references. The function call or array reference would simply be recognized as a member of a special class of "tokens". For example, adding the identifier "f" in front of an open parenthesis would cause the subexpression following the parenthesis node to be treated as the first actual parameter of the function "f." A comma (",") operator would be added to the algorithm to handle multiple parameters. A correct subtree representing the function call would be treated as a terminal node by rest of the algorithm.

The keywords of statements, such as ":=", could be treated as operators with the same special precedence as matched parentheses, easing the extension of the algorithm to statements. A difficulty arises in the case of multiple or ambiguous keywords, such as "if-then" and "if-then-else." This case could be handled by ignoring the "then" token and treating the "else" token similarly to the close parenthesis: the "else" node is migrated up the statements list to the matching "if-then" and changes the type of the node to "if-then-else." Statements and arbitrary-arity operators require that the syntax-directed editor supports lists as well as fixed-arity nodes, which is true of all editors for realistic programming languages.

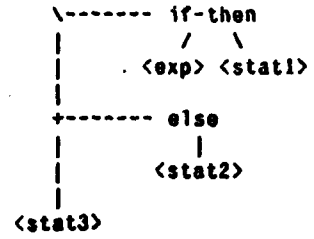


Figure 6-3: Before restructuring.

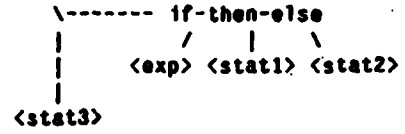


Figure 6-4: After restructuring.

Another extension to the algorithm is the capability to position the cursor at a consecutive sequence of tokens as well as a single token. The **extend-left** and **extend-right** commands would be provided for the user to stretch the cursor over adjacent tokens, one at a time. In *elastic token mode*, the editing commands would be:

- **construct:** The effect of the elastic cursor on the **construct** command is to permit a string of tokens to be sequentially inserted from the command line, either before or after the current cursor position.

- **delete:** Similarly, the effect on the **delete** command is to permit a string of tokens to be sequentially deleted.

- **clip:** **Clip** copies the tree structure representing the selected sequence of tokens to a buffer.

- **insert:** **Insert** adds the clipped structure to the syntax tree, restructuring as necessary.

Elastic mode **construct** and **delete** could be handled by repeated application of the existing algorithm (although they could be speeded up considerably by special implementation). However, **clip** and **insert** of arbitrary sequences can not currently be handled. An easy way to do the extension would be a **copy** command, which took a portion of a tree as input and copied each successive token into a buffer or at the cursor position, using the normal **construct** command to build the clipped tree or insert a clipped tree into the editing tree. The **copy** command could be implemented using the existing transformations. It is likely that some shortcut transformations could also be found.

6.3 Complexity

The algorithm is linear in time and space. The various **construct** commands are O(n + m), where n is the number of nodes on the path between where a token is inserted in the tree and the root of the tree and m is the length of the path to the right- or left-adjacent token. Only the nodes on this path, plus a constant number of other nodes adjacent to each node on this path, are considered by the **construct** algorithms. O(m) could be reduced to a constant by threading tokens according to their left-to-right and right-to-left positions in the display.

The **delete** command is O(n + m), where n is the number of nodes between the deleted token and the root or fringe of the tree, depending on the particular circumstances, and m is the length of the path between the deleted token and its left- (or right-) adjacent token in the display. This (m) must be considered since the cursor is moved to the left adjacent token by a delete. In the other commands, the cursor remains at the new node or the selected node. As above, O(m) could be replaced by a constant by threading the leaf nodes with both left-to-right and right-to-left links. This would require more space and involve modifications to the internal representation of the syntax tree.

The **replace** command is effectively a delete, without cursor motion, followed by a construct. It takes O(2n) = O(n).

The commands are implemented using the transformations presented in Chapter 5. The simple transformations, such as fill, nest, unnest and twiddle require constant time. The rippleup transformation compares the node to its ancestors and uses the macrotwiddle transformations to make the necessary changes in the tree. It looks at O(n) nodes. The rippledown transformation compares the node to its descendants and migrates the node down the tree. It looks at O(2n) = O(n) nodes. In the worst case, n is the height of the tree for rippleup and rippledown. The collapse transformation, which performs the bulk of an operator or terminal delete, requires O(m) time to find the node to which to move the cursor and O(n) time to rippleup the node that replaces the deleted node. As above, O(m) could be made constant by threading. Matchparens requires O(n) time to find the matching parenthesis, plus O(n) time for the rippleup invocations.

Thus the algorithm is linear in the depth of the expression tree. In practice, it is generally linear in the number of nodes that actually must be migrated by the transformations, but in the worst case additional nodes may be considered.

Since no auxiliary data structures are created, space is either the same as time to reflect the cost of the activation records for the recursion, or O(1) if recursion is considered free.

6.4 Implementation

We have implemented our incremental expression parsing algorithm for binary infix operators in Maclisp on Tops-20. This is a stand-alone implementation, and is not embedded in a syntax-directed editor. It provides the **construct after**, **construct before**, **construct at**, **delete** and **replace** commands and displays the tree by unparsing it into infix form. Both structure-oriented (**next**, **previous**, **in**, **out**) and display-oriented (**left**, **right**) cursor commands are supported. All trees are constructed using a small set of operators and identifiers, both of which can be extended by augmenting a table.

The complete algorithm is in the process of being implemented as part of the command interpreter of the Display Oriented Structure Editor (DOSE) system [1, 5] being developed by the Software Technology group at Siemens Corporation Corporate Research and Technology at Princeton, NJ. The initial implementation is in and for PERQ Pascal on the Three Rivers Computer Corporation PERQ personal computer.

6.5 Summary

This document describes an incremental parsing algorithm significantly different from other approaches to incremental parsing for syntax-directed editors. Our algorithm does not require maintenance of the parser state or a text buffer. It requires no data structures other than the syntax tree itself. Our algorithm does not rely on modifications to a traditional parser. It uses a small set (less than a dozen) of tree transformations that are sufficient to handle a superset of the precedence languages. All language-dependent information is contained in the operator precedence table. Our algorithm modifies the tree immediately, during each editing operation. The resulting expression can be immediately displayed by refreshing the screen. Finally, our algorithm is robust in the face of certain types of syntax errors. And, in contrast to the typical template-driven approach to syntax-directed editing, the user edits in terms of the tokens as they are displayed on the screen rather than in terms of the internal syntax tree.

Acknowledgements

We would like to thank Raul Medina-Mora and Peter Feiler for the initial presentation of the problem and for interesting us in finding a solution. The efforts of Nico Habermann, David Dill, Mike Kazar and David Notkin in reading and commenting on earlier versions of this document are greatly appreciated. We would also like to thank Kesav Nori for his comments on our approach.

References

[1] Peter H. Feiler and Gail E. Kaiser.
A Display-Oriented Structure Manipulator: A Multi-Purpose System.
January 1982.
Siemens Corporation Corporate Research and Technology. Submitted for publication.

[2] Carlo Ghezzi and Dino Mandrioli.
Incremental Parsing.
ACM Transactions on Programming Languages and Systems 1(1), July 1979.

[3] Carlo Ghezzi and Dino Mandrioli.
Augmenting Parsers to Support Incrementality.
Journal of the ACM 27(3), July 1980.

[4] A. N. Habermann and David S. Notkin.
The Gandalf Software Development Environment.
January 1982.
CMU Department of Computer Science. Submitted for publication.

[5] Gail E. Kaiser and Peter H. Feiler.
Generation of Language-Oriented Editors (Summary).
July 1982.
Siemens Corporation Corporate Research and Technology. In progress.

[6] Raul Medina-Mora and David S. Notkin.
ALOE Users' and Implementors' Guide.
Technical Report CMU-CS-81-145, CMU Department of Computer Science, November 1981.

[7] Raul Medina-Mora.
Syntax-Directed Editing: Towards Integrated Programming Environments.
PhD thesis, Carnegie-Mellon University, March 1982.

[8] M. Mikelsons and M.N. Wegman.
PDE1L: The PL1L Program Development Environment Principles of Operation.
Technical Report RC 8513, IBM T.J. Watson Research Center, October 1980.

[9] Joseph M. Morris and Mayer D. Schwartz.
The Design of a Language-Directed Editor for Block-Structured Languages.
In *Proceedings of the ACM SIGPLAN SIGOA Symposium on Text Manipulation*. ACM, June 1981.

[10] Tim Teitelbaum and Thomas Reps.
The Cornell Program Synthesizer: A Syntax-Directed Programming Environment.
Communications of the ACM 24(9), September 1981.

[11] Mark Wegman.
Parsing for Structural Editors (Extended Abstract).
In *21st Annual Symposium on Foundations of Computer Science*. IEEE, October 1980.

[12] Mark Wegman.
Details of the Wegman Algorithm.
Private communication.

[13] Mark Wegman and Cyril N. Alberga.
Parsing for a Structural Editor (part II).
Technical Report RC 9197, IBM Watson Research Center, January 1982.

I. Appendix

This appendix includes a table of operator precedences, a figure that associates each procedure with the set of transformations it uses and a pseudo-code version of the two Ripple transformations and the algorithms that implement the various user commands.

operator	left-precedence	right-precedence
+	1	1
-	1	2
*	3	3
↑	6	5

Figure I-1: Table of Operator Precedences.

	basic transformations	compound transformations	calling procedures
inserting identifiers and operators	fill nest (right/left) twiddle (right/left)	macrotwiddle (right/left)	rippleup
inserting parentheses	matchparens	macrotwiddle (open/close)	
deleting replacing and swapping	unnest collapse		rippledown

Figure I-2: Summary of the Transformations

```
procedure RippleUp (tree : Node)
var parent : Node
begin
    if ( not IsRoot(tree) and not IsLeaf(tree) ) then
        if ( IsParens(tree) ) then
            if ( not MatchParens(tree) ) then
                select ( GetParensType(tree) ) of
                    open:       MacroTwiddleOpen(tree)
                    close:      MacroTwiddleClose(tree)
                    match:      RippleUp(parent)
                endselect
        else if ( not IsMatch(parent) ) then
            select ( GetSide(tree) ) of
                right:  MacroTwiddleLeft(tree)
                left:   MacroTwiddleRight(tree)
            endselect
        endif
    endif
end


procedure RippleDown (tree : Node)
var left, right : Node
begin
    left := GetLeftChild(tree)
    right := GetRightChild(tree)
    if ( IsOp(left) ) then
        if ( LeftPrec(GetOp(tree)) > RightPrec(GetOp(left)) ) then
            TwiddleLeft(tree, left)
            RippleUp(left)
            RippleDown(tree)
        endif
    endif
    if ( IsOp(right) ) then
        if ( RightPrec(GetOp(tree)) > LeftPrec(GetOp(right)) ) then
            TwiddleRight(tree, right)
            RippleUp(right)
            RippleDown(tree)
        endif
    endif
end

procedure ConstructAfter
var token : String
    type : TypeOfToken
begin
    while ( true ) do
        read(token, type)
        select ( type ) of
        OP:     ConstrOpAfter(token)
        OPEN:   ConstrOpenAfter
        CLOSE:  ConstrCloseAfter
        TERM:   ConstrTermAfter(token)
        endselect
    endwhile
end

procedure ConstrOpAfter ( newop : String )
var temp : Node
begin
    temp := FindRightToken(Cursor)
    if ( IsEmptyOp(temp) ) then
        FillOp(temp, newop)
        Cursor := temp
        RippleUp(Cursor)
```

```
    else
        select ( GetType(Cursor) ) of
        OP, OPEN:
            NestRight(temp, newop)
            Cursor := GetParent(temp)
            RippleUp(Cursor)
        TERM, CLOSE:
            NestLeft(Cursor, newop)
            Cursor := GetParent(Cursor)
            RippleUp(Cursor)
        endselect
    endif
end

procedure ConstrTermAfter ( newid : String )
var temp : Node
begin
    select ( GetType(Cursor) ) of
    OP, OPEN:
        temp := FindRightToken(Cursor)
        if ( not IsMeta(temp) ) then
            NestRight(temp, '$OP')
            temp := GetLeftChild(GetParent(temp))
        endif
        Cursor := temp
        FillMeta(Cursor, newuid)
    TERM, CLOSE:
        NestLeft(Cursor, '$OP')
        temp := GetRightChild(GetParent(Cursor))
        Cursor := temp
        FillMeta(Cursor, newid)
    endselect
end

procedure ConstrOpenAfter
var temp : Node
begin
    select ( GetType(Cursor) ) of
    OP, OPEN:
        temp := FindRightToken(Cursor)
        NestRight(temp, '(')
        Cursor := GetParent(temp)
        RippleUp(Cursor)
    TERM, CLOSE:
        error
    endselect
end

procedure ConstrCloseAfter
begin
    select ( GetType(Cursor) ) of
    OP, OPEN:
        error
    TERM, CLOSE:
        NestLeft(Cursor, ')')
        Cursor := GetParent(Cursor)
        RippleUp(Cursor)
    endselect
end


procedure ConstructBefore
var token : String
    type : TypeOfToken
begin
    read(token, type)
```

```
    select ( type ) of
    OP:         ConstrOpBefore(token)
    OPEN:       ConstrOpenBefore
    CLOSE:      ConstrCloseBefore
    TERM:       ConstrTermBefore(token)
    endselect
end

procedure ConstrOpBefore ( newop : String )
var temp : Node
begin
    temp := FindLeftToken(Cursor)
    if ( IsEmptyOp(temp) ) then
        FillOp(temp, newop)
        Cursor := temp
        RippleUp(Cursor)
    else
        select ( GetType(Cursor) ) of
        OP, CLOSE:
            NestLeft(temp, newop)
            Cursor := GetParent(temp)
            RippleUp(Cursor)
        TERM, OPEN:
            NestRight(Cursor, newop)
            Cursor := GetParent(Cursor)
            RippleUp(Cursor)
        endselect
    endif
end

procedure ConstrTermBefore ( newid : String )
var temp : Node
begin
    select ( GetType(Cursor) ) of
    OP, CLOSE:
        temp := FindLeftToken(Cursor)
        if ( not IsMeta(temp) ) then
            NestLeft(temp, '$OP')
            temp := GetRight(GetParent(temp))
        endif
        Cursor := temp
        FillMeta(Cursor, newuid)
    TERM, OPEN:
        NestRight(Cursor, '$OP')
        temp := GetLeftChild(GetParent(Cursor))
        Cursor := temp
        FillMeta(Cursor, newid)
    endselect
end

procedure ConstrOpenBefore
var temp : Node
begin
    select ( GotType(Cursor) ) of
    OP, CLOSE:
        error
    TERM, OPEN:
        NestRight(Cursor, '(')
        Cursor := GetParent(Cursor)
        RippleUp(Cursor)
    endselect
end

procedure ConstrCloseBefore
begin
    select ( GetType(Cursor) ) of
```

```
    OP, CLOSE:
        temp := FindLeftToken(Cursor)
        NestLeft(temp, ')')
        Cursor := GetParent(temp)
        RippleUp(Cursor)
    TERM, OPEN:
        error
    endselect
end


procedure ConstructAt
var token : String
    type : TypeOfToken
begin
    if ( IsEmptyOp(Cursor) ) then
        read(token, type)
        if ( type = OP ) then
            FillOp(Cursor, token)
            RippleUp(Cursor)
        else
            error
    else if ( IsMeta(Cursor) ) then
        read(token, type)
        select ( type ) of
        TERM:
            FillMeta(token)
        OP, CLOSE:
            NestLeft(Cursor, token)
            RippleUp(token)
        OPEN:
            NestRight(Cursor, token)
            RippleUp(token)
        endselect
    else
        error
    endif
end


procedure Delete
var node : Node
begin
    node := Cursor
    MoveCursorLeft
    select ( GetType(node) ) of
    OP:         DeleteOp(node)
    OPEN:       DeleteOpen(node)
    CLOSE:      DeleteClose(node)
    TERM:       DeleteTerm(node)
    endselect
end

procedure DeleteOp ( node : Node )
var temp : Node
begin
    temp := FindRightToken(node)
    if ( IsMeta(temp) ) then
        CollapseLeft(node, temp)
    else
        temp := FindLeftToken
        if ( IsMeta(temp) ) then
            CollapseRight(node, temp)
        else
            PutOp(node, '$OP')
            RippleDown(node)
```

```
        endif
    endif
end

procedure DeleteTerm ( node : Node )
var temp : Node
begin
    temp := FindRightToken(node)
    if ( IsEmptyOp(temp) ) then
        CollapseRight(temp, node)
    else
        temp := FindLeftToken(node)
        if ( IsEmptyOp(temp) ) then
            CollapseLeft(temp, node)
        else
            FillMeta(node, '$META')
        endif
    endif
end

procedure DeleteOpen ( node : Node )
var child : Node
begin
    child := GetChild(node)
    if ( IsMatchParen(node) ) then
        PutParen(node, ')')
        RippleUp(node)
        RippleUp(child)
    else
        UnNestRight(node)
        RippleUp(child)
    endif
end

procedure DeleteClose
var child : Node
begin
    child := GetChild(node)
    if ( IsMatchParen(node) ) then
        PutParen(node, '(')
        RippleUp(node)
        RippleUp(child)
    else
        UnNestLeft(node)
        RippleUp(child)
    endif
end


procedure ReplaceOp
var newop : String
    type : TypeOfToken
begin
    read(newop, type)
    if ( (type = OP) AND (GetType(Cursor) = OP) ) then
        PutOp(Cursor, newop)
        RippleUp(Cursor)
        RippleDown(Cursor)
    else
        error
    endif
end
```